AD A042370

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

12

MIT/LCS/TR-182

A FRAMEWORK FOR PROCESSING DIALOGUE

Gretchen P. Brown

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/TR-182	2. GOVT ACCESSION NO.
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A Framework for Processing Dialogue,	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TR-182
7. AUTHOR(s): Gretchen P. Brown	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Ma 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, Department of Defense, 1400 Wilson Boulevard, Arlington, Virginia 22209	12. REPORT DATE: June 1977
	13. NUMBER OF PAGES: 82
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Department of the Navy, Information Systems Program, Arlington, Virginia 22217	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

6 14 10 15 11 (12) 76p.

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

natural language
dialogue
English understanding
English generation
OWL-1
semantics
pragmatics

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes a framework for handling mixed-initiative English dialogue in a console session environment, with emphasis on recognition. Within this framework, both linguistic and non-linguistic activities are modelled by structures called 'methods', which are a declarative form of procedural knowledge. The ~~Our~~ design focuses on units of linguistic activity larger than the speech act, so that the pragmatic and semantic context of an utterance can be used to guide its interpretation. Also important is the treatment of indirect speech acts, e.g., the different ways to ask a → (cont on p. 1473)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

409 648 LB

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

cont fr p 1473A)

20. question, give a command, etc.

Given the static model of dialogue embodied in the methods, the problem is to find the correct method step that relates to a particular input. ~~We handle~~ this problem is handled through a combination of careful structural distinctions and the use of multiple recognition strategies. The dialogue methods are used to generate expectations dynamically, special structures are used to facilitate matching, and a basic distinction between four major utterance classes is used to determine which of several overall matching strategies should be used for a given expectation.

MIT/LCS/TR-182

A Framework for Processing Dialogue

Gretchen P. Brown

June 1977

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract Number N00014-75-C-0661.

Massachusetts Institute of Technology

Laboratory for Computer Science

(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

Abstract

This report describes a framework for handling mixed-initiative English dialogue in a console session environment, with emphasis on recognition. Within this framework, both linguistic and non-linguistic activities are modelled by structures called *methods*, which are a declarative form of procedural knowledge. Our design focuses on units of linguistic activity larger than the speech act, so that the pragmatic and semantic context of an utterance can be used to guide its interpretation. Also important is the treatment of indirect speech acts, e.g., the different ways to ask a question, give a command, etc.

Given the static model of dialogue embodied in the methods, the problem is to find the correct method step that relates to a particular input. We handle this problem through a combination of careful structural distinctions and the use of multiple recognition strategies. The dialogue methods are used to generate expectations dynamically, special structures are used to facilitate matching, and a basic distinction between four major utterance classes is used to determine which of several overall matching strategies should be used for a given expectation.

Acknowledgements

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract Number N00014-75-C-0661. The specifications for Susie Software were developed by Professor William Martin, and many of the ideas presented here grew out of discussions with him and with the L.C.S. Knowledge Based Systems Group. My thanks to B. Bruce, C. Bullwinkle, L. Hawkinson, R. Krumland, E. Lewis, W. Long, W. Mark, A. Sunguroff, and P. Szolovits for their help and advice.

A Framework for Processing Dialogue

Contents

I. Introduction

In recent years, increasing attention has been given to expert systems in domains such as management information, medical records and diagnosis, algebraic manipulation, and automatic programming. The expert system has some specialized knowledge and capabilities, and its user is usually assumed to be knowledgeable in the area of application, although not necessarily knowledgeable about computer operations or computer languages. There is a good deal of support for the premise that a flexible, reliable natural language interface would increase the usefulness of expert systems and widen their user community. The task of constructing such an interface, however, has turned out to be a difficult one.

This paper describes a framework for such an English language interface, using automatic programming as the area of system expertise. The user would come to the system with an idea of the input/output behavior he or she wanted from a computer program, and the expert system would produce the program to these specifications. Automatic programming is a good domain for the purpose of exploring user/expert system interfaces because it demands relatively complex linguistic capabilities: in the normal course of writing a program not only are questions asked and answered, but descriptions and explanations must be generated as well. Any natural language interface for this domain, then, is forced to deal not only with individual sentences but also with several sentences relating to each other and to the rest of the dialogue.

The framework that will be presented is designed for typed, rather than spoken, English dialogue in a mixed-initiative console session environment. The basic approach taken has been to model dialogue as a process. Knowledge of how to carry out a dialogue is represented

explicitly in the system, primarily in the form of structures called *methods*. Methods are written in OWL, a language for representing knowledge; their function is to provide a declarative representation for procedural knowledge. Methods are used to represent both linguistic and non-linguistic activities. In addition, they are used to represent both knowledge about particular tasks (such as writing a program for someone else) and more general knowledge about dialogue (for example, the sequences of activity that correspond to asking and answering a question, asking for and giving a description, and so forth). The similar treatment of task knowledge and general dialogue knowledge stems from the conviction that although the two areas differ in some respects, activities from both areas contribute directly to dialogue production.

The introduction of a set of methods is not, however, enough. We also need mechanisms that enable us to use methods to model *particular* dialogues. There are two sides to this problem: a system must be able to relate a user input to the ongoing dialogue, and it must be able to generate an output. This paper will focus on the former requirement, which amounts to language understanding. We do not underestimate the difficulty of language generation, but it appears to be easier to achieve a minimal level of usefulness in a generator than in an understanding module.

The ideas presented here have been implemented in a prototype system called Susie Software.[1] The system currently performs the internal manipulations necessary to support the

1. Susie Software is embedded in the OWL-I system, which runs on a PDP-10 under ITS. OWL is being developed at M.I.T's Laboratory for Computer Science by the Knowledge Based Systems Group under the direction of William A. Martin. Development of the Susie Software dialogue facility began in 1974.

sample dialogue given in Figure 1. There and in other examples of dialogue, asterisks indicate lines typed by the user. Note that the prototype system currently inputs and outputs OWL expressions rather than English ones; the boundaries of the implementation will be discussed further in the course of the presentation.

In terms of current artificial intelligence research, OWL methods correspond roughly to Minsky's frames [11] or Schank and Abelson's scripts [14], although the highly centralized control structure of the OWL Interpreter differentiates it from many existing implementations. The dialogue environment that we have chosen is similar to that described by Deutsch [5] for the workstation domain and also that described by J.S. Brown et.al. [2] for the SOPHIE system, but our work differs from both of these in the incorporation of an explicit model of dialogue activities. Finally, the methods that are used to model dialogue resemble the social action paradigms of Bruce [3] as well as the representations of dialogue games described by Levin and Moore [8]. The most important contrast between methods on the one hand and social action paradigms and dialogue games on the other is the distinction between standard path method steps and recovery path steps. This distinction, which is strongly motivated by access considerations, will be discussed in more detail below.

In the sections that follow, we look first at the ways that the Susie Software environment was constrained in an effort to limit the problem. Section 3 then discusses the use of methods to model dialogue activities, and Section 4 gives a brief introduction to the OWL-I system. Focussing on recognition, Section 5 summarizes the approach taken, and sections 6 through 9 discuss recognition for individual utterance types. Finally, Section 10 considers the prototype system in the light of the requirements of a practical working environment. An introduction to OWL-I notation can be found in the appendix.

Hello, I am Susie. What is your name?
◦ Bill.
◦ I need a program to manipulate blocks written.

I can help you write a program which builds a scene using the blocks from another scene.
◦ That is what I want.

O.K. I need a description of the initial and final scenes.
◦ The final scene contains 3 blocks: B1, B2, and B3.

O.K.
◦ What else do you want to know about the final scene?

What are the positions of B1, B2, and B3 with respect to each other?
◦ B1 is on top of B2 and B3 is on top of B1.

O.K.
◦ Initially, all of them are on the table.

O.K.
◦ Can you write the program from that?

I have enough information to start writing the program. I can't determine
all of the information which I will need until I write the program.
◦ O.K.

Do you want me to decide implementation issues such as what hand to grasp a block with?
◦ Yes.

I have finished the program. Do you want me to describe it?
◦ Yes.

Using my left hand, first I put B1 on top of B2, and then I put
B3 on top of B1.
◦ Good-bye.

Figure 1. A sample dialogue with Susie Software.

2. Keeping Dialogue Manageable

In the Susie Software effort, we have explored ways to achieve a level of dialogue flexibility without sacrificing processing reliability and while keeping response time within reasonable bounds. Unfortunately, the experience in computational linguistics has been that even relatively small amounts of flexibility in a natural language system can lead to grave combinatorial difficulties. To keep the dialogue problem manageable, Susie's environment was carefully constrained along the following dimensions:

(1) communication channel
(2) choice of application area
(3) number of tasks performed
(4) complexity of semantic domain

Along the first of these dimensions, communication channel, the limitation to typed input and output has already been mentioned. We are also assuming that input is free of spelling errors and gross grammatical inaccuracies (although full sentences are not required). Finally, there is a slight control bias on the side of the system, since the user may type only when an asterisk is output by Susie and then only a single sentence or sentence fragment at a time.

The second constraint is the choice of application area. Here, the decision to construct a task-oriented environment (in the sense used by Deutsch [5]) as opposed to, say, a system to model casual dialogue already acts as a constraint. For example, a task environment defines the aspects of user input that are important, so that the problem of deciding why a piece of information was input is greatly reduced. Beyond this, the structure of the primary task area chosen for Susie Software -- programming -- allows us to exploit strong expectations once a

specific task is underway. At any given point in the course of producing a program there are only a small number of basic ways that the user and the system may interact (although the *content* of these interactions, of course, will vary considerably).

Third on the list of constraints is the number of different tasks performed. Susie's projected abilities are limited to writing programs and answering user questions about system capabilities and previous activities. This means that the range of user requests for new tasks is relatively small. The limitation on tasks turns out to be particularly helpful in a mixed-initiative environment where any user utterance may theoretically be the start of a new task.

The fourth and final constraint is on the scope of the semantic domain. We have been working with the world of 2-dimensional toy blocks, a common starting point due to the simplicity of the domain and its clearly defined semantics. We envision a system built along the principles found in Susie Software to be useful in areas such as business payroll processing, inventory control, etc. These domains are not trivial ones, but they also are not as complex as some others.

Besides constraining the environment in the four ways discussed, we believe it is necessary to embed a significant amount of knowledge in a system. This knowledge is of two basic types: knowledge about the semantic domain and knowledge about dialogue in general. Taking the semantic domain first, we feel that it is essential that the domain be spanned in order for a system of this type to be practical. By "spanned" we do not necessarily mean that every possible user question can be answered or every possible program produced. Instead, we mean that the system should have a good enough model of itself so that questions or requests that cannot be handled can be given appropriate responses. A simple "I don't understand"

would not be considered adequate in most cases. Note that at present no attempt is made to span the semantics of the blocks world, and correspondingly the implementation lacks robustness; the domain-specific knowledge that is in the system, however, seems to be a representative sample.

In addition to knowledge about the semantic domain, we mentioned knowledge about dialogue. We believe that a flexible, reliable, and efficient dialogue processing system needs a sound model of dialogue structure. In a sense, this model is the grammar of dialogue, although the model that will be discussed here is a computational one, not easily reduced to a set of rules. (No such attempt has, in fact, been made.) The term *structure* is used for what others might call the syntax of dialogue. Many "structural" phenomena are semantic in flavor (although they do not necessarily vary according to the specific semantic domain), and the use of the term *syntax* might be misleading. For example, the fact that questions get answers is easily enough called "dialogue syntactic," but the fact that answers may have associated stipulations ("Yes, if...") or qualifications ("Yes, but...") begins to stretch the connotations of the term *syntactic*. The model of dialogue to be discussed will therefore be referred to as a structural one.

The Susie Software environment, then, has two properties that we consider crucial for a first pass at flexible, practical, man-machine dialogue. First, it is carefully constrained, and, second, the system contains a model of the structural aspects of dialogue. It is this structural model that will be the topic of the rest of this paper.

3. Conversational Exchanges

3.1. What They are and How to Model Them

Looking at the sample dialogue, we can pick out groups of lines that appear to belong together. A simple example of such a unit would be a question plus its answer:

What is your name?
◆ Bill

This is an example of what we will call a *conversational exchange.* At the more complex end of the conversational exchange scale (although outside the scope of the Susie system), would be an exchange like formal debate. A debate program might consist of several subsections and many steps. The conversational exchange is a structure at the interpersonal level of dialogue. This places it at a higher level than the speech act (Searle [16]), which does exist in a conversational environment but which is carried out by a single agent. For example, promising and accepting the promise would be two speech acts, but they would form a single conversational exchange. The notion of a conversational exchange is similar in spirit to the ideas of sequence and adjacency pair formulated by Schegloff and Sacks [15]. In both cases the concern is to establish a structural unit with associated expectations, so that it is possible, among other things, to account for a situation where some conversational event does *not* occur.

This is not to say, of course, that dialogues can only proceed by relentlessly completing conversational exchanges. Many dialogues are much less orderly, their structure influenced by a set of competing goals. In Carbonell [4] we see an example of a conversational exchange (a question-answer sequence) that remains temporarily uncompleted:

Approx what is the area of Argentina?

Tell me something about Peru

Peru is a country.
It is located in South America.
The capital of Peru is Lima.

Now answer the question you didn't answer before.

The area of Argentina is 1,100,000 square miles.

Thus, while the notion of a conversational exchange is an important one, it should not be rigidly applied in modelling a particular dialogue. We will see that the Susie Software system uses the expectations set up by a conversational exchange in attempting to understand a user input, but these expectations are used in a very flexible way.

Conversational exchanges are modelled using OWL-I methods. Since the question-answer exchange is a relatively simple example, we will pursue it further. An English translation of the OWL-I method for the question-answer exchange is given in Figure 2. The actual method, along with an introduction to OWL-I notation, can be found in the appendix. Methods have three main parts: a header, argument specifications, and procedure steps. The header is the method's unique name, the argument specifications, organized into semantic cases, are used for type checking of inputs to the method or to specify the form of outputs, and procedure steps, along with their associated input case assignments, form the body of the method. The steps come in two varieties, *standard path* and *recovery path*. This distinction will be discussed further later on, but, basically, standard paths represent the ways that an exchange can "go right," while recovery paths give some of the possible measures to be taken when an exchange gets "off the track."

The structure of OWL-I methods is reflected in Figure 2. Here, "ask-and-answer" corresponds to the header of the actual method, ASK-AND-ANSWER, which is an OWL-I concept. (Concepts, the basic unit of OWL-I, will be discussed in Section 4; here and throughout this paper OWL-I concepts will appear in capital letters.) The argument specifications for this example consist of input cases only; output case specifications describing the results of methods do occur frequently, but none were used for ASK-AND-ANSWER. Looking at the object case specification in Figure 2, we see that ASK-AND-ANSWER handles all questions besides how- and why-questions. These other two varieties of question are handled by ASK-AND-DESCRIBE and ASK-AND-EXPLAIN, two methods that will be discussed further later on in this section. Two other semantic cases used for ask-and-answer are the agent and co-agent, corresponding to the participants in the exchange. By convention, the agent of an entire dialogue method is the agent of the first step, so that we can identify the agent of the method with the participant who starts it off.

The standard path steps for the question-answer exchange are steps 1 though 4 in Figure 2. The ask-and-answer method also has two recovery paths, although more could be written easily enough. The first handles the situation where an answer can only be given if there is an associated stipulation, e.g., if line 19 of the sample dialogue were, "Yes *if I can ask you about them later?*" The second recovery path handles the case where no answer can be found. Both of these failures occur in the process of finding the answer, step 3, but the recovery path is associated with the ask-and-answer method. A different recovery path would be used if the find-answer routine were called in another context, e.g., reasoning.

An important point about the ask-and-answer example is that it contains the parts played

ask-and-answer

object: the question to be asked
(not a how- or why- question)
agent: a person or computer system
co-agent: a person or computer system

method:

1. *The agent asks the question.*
2. The co-agent now knows what the question is.
3. The co-agent finds the answer.
4. The co-agent gives the answer and the agent gives an (optional) acknowledgement.

recovery path 1: if a stipulation is found along with the answer
R1.1 The co-agent states the stipulation.
R1.2 The agent agrees to it.

recovery path 2: if the answer is unknown
R2.1 The co-agent says that he doesn't know the answer.

Figure 2. An English representation of a method for asking a question and getting an answer.

by both speakers and is intended for use by the Interpreter whether it is Susie or the user who is the one to initiate the exchange by asking the question. we will call this latter property *speaker independence*. If ask-and-answer and the other dialogue methods are speaker independent, then it is up to the Interpreter to determine whether a particular utterance is to be generated or understood. There are, in fact, three possible modes of interpretation[2] for a step in a dialogue method:

(1) Carry out the step (e.g., ask a question).

(2) Recognize that a step has happened (e.g., that an answer to your question has been given).

(3) Assume that a step has happened (e.g., if your conversational partner gave the answer, then he had to perform the mental process of finding the answer first.)

Given the input case settings in a call, the Interpreter uses a set of simple rules to determine the mode of a step. Modelling the actions of both participants in a single method simplifies programming, reduces the total number of methods to be written and maintained, and increases the usefulness of methods, since they may also be used in a pure recognition situation, where the hearer is not a participant in the dialogue.

One final general point about the dialogue methods is their abbreviated form. For any given speech act, only production is represented explicitly, and the activities of the other

---

2. A difference of terminology here. Throughout this paper, the words *interpret* and *interpretation* will be used to indicate actions of the OWL-I Interpreter. The meaning of *interpret* found in *natural language interpretation* (as opposed to *natural language generation*) will be conveyed by *recognize* and its variants.

partner are left implicit. Thus, although the Susie programs are speaker independent in the technical sense defined, they are not without a bias: it does not matter who is specified as the agent of a communication step, but, whoever is, the "story" is told from his point of view.[3] "Listening" steps are left implicit not because they are unimportant, but because the form and timing are predictable. Where a joint model of communication was necessary (e.g., when misunderstandings occur) the system would be expected to expand the abbreviated model expressed in the dialogue methods.

3.2. Core Methods

In addition to the ask-and-answer method, the Susie Software system currently contains thirty other methods needed either to run the sample dialogue directly or to provide a rich enough environment to test the procedure selection and matching routines. Recall that the method representation conventions do not require the OWL-I programmer to distinguish between methods designed solely for dialogue, methods that use dialogue to gather information in order to get other work done, and non-dialogue methods such as block manipulation routines. It is useful, however, to distinguish domain dependent methods that will necessarily grow as the system is extended from the set of domain independent methods that can be programmed once and for all, requiring only minor modifications thereafter. This domain independent group of methods which consists of dual-participant methods built around speech acts, will be called *core dialogue methods*.

---

3. In describing the "active" bias of the method representation style, we do not mean to imply that the rest of the system shares this bias. In fact, it does not.

Since core dialogue methods are organized around speech acts, let us start with the treatment of speech acts in the Susie Software system. The current implementation distinguishes three categories which are represented by their corresponding OWL-I concepts as COMMAND-REQUEST, ASK, and TELL. COMMAND-REQUEST encompasses the full range of requests for a nonverbal activity, ranging over the different authority relationships from ordering to pleading. Besides nonverbal activities, it also handles requests for speech acts for which the requester is not to be the destination (e.g., "Tell Harry what you told me."). ASK conveys a request for information, and TELL is the act that conveys information. This is a very simple taxonomy, and as a system became more complex we would want to see it enriched. Searle, in [17], suggests five categories that would be useful as a top level of speech act organization for implementation: representatives (to commit the speaker to the truth of a proposition), directives (to get someone to do something, either verbal or nonverbal), commissives (to commit the speaker to some future course of action), expressives (e.g., thank, apologize, welcome), and declaratives (institutionalized speech acts such as christening a ship or declaring war). These categories are useful, but it is the next, more specialized, level in the taxonomy that is of most interest to us here, since this is the level that would begin to have corresponding core dialogue methods. Let us consider each of Searle's classes in turn, looking at the class members important to the Susie Software implementation and their corresponding core methods.

In a task-oriented system, the representative of most interest is TELL, since this is the speech act that makes statements. The corresponding core method is STATE-AND-ACKNOWLEDGE, which is a TELL activity followed by an optional acknowledgement on

the part of the hearer. Moving on to directives, three varieties are of interest: SUGGEST, COMMAND-REQUEST, and ASK. SUGGEST conveys a request that the hearer entertain an idea (and so this type of request must also be ruled out of the range of COMMAND-REQUEST). SUGGEST and its corresponding core method SUGGEST-AND-ACCEPT are not used by the implementation at this time.

Two important core methods associated with COMMAND-REQUEST are COMMAND-AND-RESPOND and ASK-FOR-AND-HELP. Each starts with a COMMAND-REQUEST for an activity within the range of this speech act. The difference between the methods is that in COMMAND-AND-RESPOND the requester does not expect to do any of the task, while in ASK-FOR-AND-HELP the task is divided up between the requester (agent) and the hearer (co-agent). Thus, ASK-FOR-AND-HELP has an explicit substep for dividing up the task appropriately.

The third directive, ASK, has three associated core methods, ASK-AND-ANSWER, ASK-AND-DESCRIBE, and ASK-AND-EXPLAIN. As mentioned above, ASK-AND-ANSWER handles most what-, where-, whether-, and when-questions. Why-questions are handled by ASK-AND-EXPLAIN, and how-questions are split between ASK-AND-DESCRIBE and ASK-AND-EXPLAIN depending on the type of information that seems appropriate. Of course ASK-AND-DESCRIBE and ASK-AND-EXPLAIN can also be triggered by a direct request for a description or explanation, respectively. The motivation for distinguishing ASK-AND-DESCRIBE and ASK-AND-EXPLAIN from ASK-AND-ANSWER is that the first two will tend to be involved with longer answers that require more selection and organization of the information. ASK-AND-DESCRIBE and ASK-AND-EXPLAIN are

distinguished from each other by the aspects of the topic that are considered relevant; for ASK-AND-EXPLAIN, the emphasis is on causal relationships.

This accounts for the core methods used in the current implementation, although it is easy enough to write additional ones, such as PROMISE-AND-ACCEPT to correspond to the commissive PROMISE. In addition, we would probably want to add separate concepts and core methods for expressives such as GREET and GREET-AND-RESPOND, TAKE-LEAVE and TAKE-LEAVE-AND-RESPOND, APOLOGIZE-FOR and APOLOGIZE-AND-RESPOND, and THANK-FOR and THANK-FOR-AND-ACKNOWLEDGE. All of these speech acts are used in the implementation, although they are currently handled by TELL, with any response also conveyed using a TELL. Searle's fifth class, declaratives, seems to be less relevant for a console session environment.

The core methods described constitute a good working set for the Susie Software environment. The reader should keep in mind, however, that core methods are only part of the total set of dialogue methods. Later on in this section, core methods and other methods will be combined to analyze the sample dialogue.

3.3. Basic Utterance Types

In analyzing dialogues, it is useful to distinguish four basic types of utterance. Several of these have been mentioned previously, but it is time to give them a more formal introduction. Viewed in terms of methods, the first two categories of utterance correspond to types of method steps, and the third category contains deviations from the normally expected path or paths,

with the deviations, again, having corresponding method steps. The fourth category contains utterances that specify how other utterances fit into the ongoing dialogue.

In a task environment the tasks must begin in some way, and we will call the class of utterances that may start off a task *initiators*. Recall that in the Susie Software environment there are two types of tasks, program writing and question answering. (Each of these may of course be composed of subtasks.) In this environment, then, initiators are either requests for a program or for information. Not all conversational exchanges are started off by initiators. An utterance corresponding to the first step in STATE-AND-ACKNOWLEDGE, for example, will be called a lead-in; it is not an initiator, however, since it would not start off one of Susie's two top level activities. Note that initiator and lead-in utterances are not necessarily produced by the first steps of methods; instead, they are produced by the first speech act steps. A mental process step, for example, might be the first step in a method but would not produce any verbal output.

The next utterance type is the standard path successor step: an utterance may correspond to a step in a task that is already underway. Examples would be the answer to a question or the acknowledgement of a statement. Standard paths are the normally expected sequences of events for an activity. When a description of an activity is given, it is the standard path steps that are included. If deviations from the standard path(s) are described at all, it is only the most important ones that are given. This indicates that the standard paths of an activity embody the minimum of necessary information about it. Further justification of the notion of standard path is found in [1].

A third basic utterance type is *recovery discussion*. The standard paths of a dialogue

method are intended to specify the relatively small number of ways that an exchange can be concluded successfully. This is fine as long as the dialogue goes as intended and no expectations are violated. In practice this will probably not be long, and recovery discussion will result. Let us look at an example from the sample dialogue:

6 O.K. I need a description of the initial and final scenes.
* The final scene contains 3 blocks: B1, B2, and B3.
O.K.
* What else do you want to know about the final scene?
10 What are the positions of B1, B2, and B3 with respect to each other?
* B1 is on top of B2 and B3 is on top of B1.
O.K.
* Initially, all of them are on the table.

In line 6 Susie asks for a description and in line 7 the user starts to give it. This is represented by the method ASK-AND-DESCRIBE. At line 9 the user indicates that his model of what Susie wants to know is insufficient. When this happens, a recovery path is entered to accomplish the same goal by assuming less knowledge on the part of the user. This is reflected in line 10, where Susie asks a question, thereby communicating what it is she wants to know. By line 12, the difficulty has been cleared up, and the dialogue is back on the standard path of the ask-and-describe exchange.

Among the failure conditions that will generate discussion, we have concentrated on lack of the information necessary to make a decision, since this is the case that comes up in the Susie dialogue. Other recovery discussion may come up as a result of contradictions (in a sense, the overabundance of information) and misunderstandings. Of the structures needed to model recovery discussion, one, the recovery path, has already been introduced. Recovery paths are a

very local way to model recovery discussion, and they are not expected to be useful for all cases where expectations are violated. A more general mechanism is also necessary, and for this we look to autonomous OWL-I methods for handling particular failures. Such autonomous methods are part of the general OWL-I failure mechanism. Note that the sample dialogue does not contain any lines that have been modelled with the general failure mechanism, so that this possibility will not be considered in detail in this paper.

Turning from recovery discussion, the fourth basic utterance type is *metadiscussion*. Utterances classified as metadiscussion deal with the conversational situation itself. These utterances are used to change the flow of activity in a dialogue or clarify the current flow of activity. Based on the dialogues we have looked at, it appears that true metadiscussion involves a relatively narrow range of utterances. Many utterances that one would initially class as metadiscussion because they deal with the conditions of conversation turn out, on closer examination, to be better classified as recovery discussion. For the Susie Software environment, we have found only three categories of utterances that are purely metadiscussion. The user can either suspend an activity ("Let's stop this for now."), reopen a suspended or closed one ("I want to go back to the first program you wrote."), or specify what he or she is going to do next ("Now I'll tell you about the final scene."). Note that the sample dialogue does not contain any examples of metadiscussion as we have defined it, although metadiscussion has been handled in the design of the recognition process.

This finishes the discussion of the four basic utterance types: initiator, standard path successor step, recovery discussion, and metadiscussion. The distinction will be important to the system when it comes time to develop structural expectations about the form of a user input, and this distinction will form the basis for the approach to matching for recognition.

3.4. Analyzing the Sample Dialogue

Given the dialogue methods and the four basic utterance types, we can describe the conversational exchange structure of the sample dialogue from Figure 1. The sample dialogue is first of all a console session, so we have the method ((PARTICIPATE IN) CONSOLE-SESSION). (The significance of parentheses in OWL-I concepts is explained in the appendix.) The first steps of this procedure handle the greeting and introductions (lines 1-2) and the last step handles the closing (line 23). In the middle of this method is the call to carry out one of Susie's two top level activities, writing a program or answering a question, a step that may be repeated an indefinite number of times. The sample dialogue shows one of the top level activities: lines 3-22 contain a program writing exchange.

We turn now to the program writing exchange, with line 3 as the initiator. This line triggers COMMAND-AND-RESPOND. In lines 4-5 Susie finds that her capabilities are not as broad as the user's general request, and an attempt is made to get a more specific idea of what the user wants. These lines are treated as a temporary departure from the COMMAND-AND-RESPOND method onto a recovery path.

Once the user's request is clarified, the system enters the (WRITE PROGRAM) method. In this procedure, conversational steps are intermixed with non-conversational ones, i.e. the actual program-writing calls. Susie's first step in writing a program is a call to (GET DESCRIPTION), where there are currently two alternatives. If the user probably has no idea of the properties of the input and output that are of interest, he or she can be guided through the description by a series of ASK-AND-ANSWERs for which Susie generates questions. If the

user is assumed to know the relevant aspects, as is the case in the sample dialogue, then a subcall is made to ASK-AND-DESCRIBE. In the sample, the request and the subsequent description constitute the exchanges from lines 6 to 17. In the course of the description the user finds he does not know exactly what Susie wants to know, and a recovery path is entered, as discussed previously. This is done instead of a more general reselection of strategies, which is a more difficult process to control. When the difficulty is past, the dialogue returns to a normal description-giving process, which is brought to a close (implicitly) by the user question and its answer, lines 15-17.

With input and output conditions described, Susie can now go on to write the program. More information is needed, however, so she returns to the user with a question-answer exchange handled by ASK-AND-ANSWER, lines 18-19. When the program is finished, Susie notifies the user and then does an ASK-AND-ANSWER to find out whether a description of the program is wanted (lines 20-21). Since in this case the user does want a description, that becomes the final step of the (WRITE PROGRAM) method (line 22). At the same time, this utterance completes the open COMMAND-AND-RESPOND activity and the activity for achieving a top level task.

We have seen how the idea of a conversational exchange and the corresponding OWL methods can be used to analyze the structure of a dialogue. This is only part of the story, however, since mechanisms are needed to choose the particular methods that model each line. These mechanisms will be introduced in the sections that follow.

4. An Introduction to the OWL System

The Susie Software system is embedded in OWL-I. A general understanding of the aims and operation of the OWL-I system is necessary for an understanding of the recognition mechanisms that will be described, so a brief introduction will be given here. OWL-I became operational in September 1975; since that time, a new version of the system, OWL-II has been under development. The OWL system is continually evolving toward two goals: first, to provide an environment for the representation and use of expert knowledge and, second, to do limited-domain processing of natural language. It is our belief that the paths to meet these two goals are not completely disjoint and that some of the organizational principles and structures used to handle English will carry over to the structuring and use of expert knowledge as well. This sections surveys the major modules of the OWL-I system; see the appendix for an explanation of OWL-I notation.

Figure 3 shows the major modules of the OWL-I system.[4] The Linguistic Memory System (LMS) is described in [7] and [18], and is used to build and maintain the knowledge base. For our purposes here, it is sufficient to say that OWL representations are made up of data structures called *concepts*, using the operations of *specialization* and *modification*. Specialization is the subcategorization operation used for hierarchical ordering of concepts. Modification allows properties, including complex structures such as procedures, to be associated with concepts. Both the immediate specializations (called *branches*) of a concept and its properties

---

4. LMS was implemented by Lowell Hawkinson, and implementations of the Interpreter's Carry-out, Evaluate, and Whether modules were done by Alexander Sunguroff, William Long, and William Swartout respectively. The generator was done by William Swartout, and the parser is currently being developed by William Martin and Peter Szolvits. Other modules were implemented by the author.

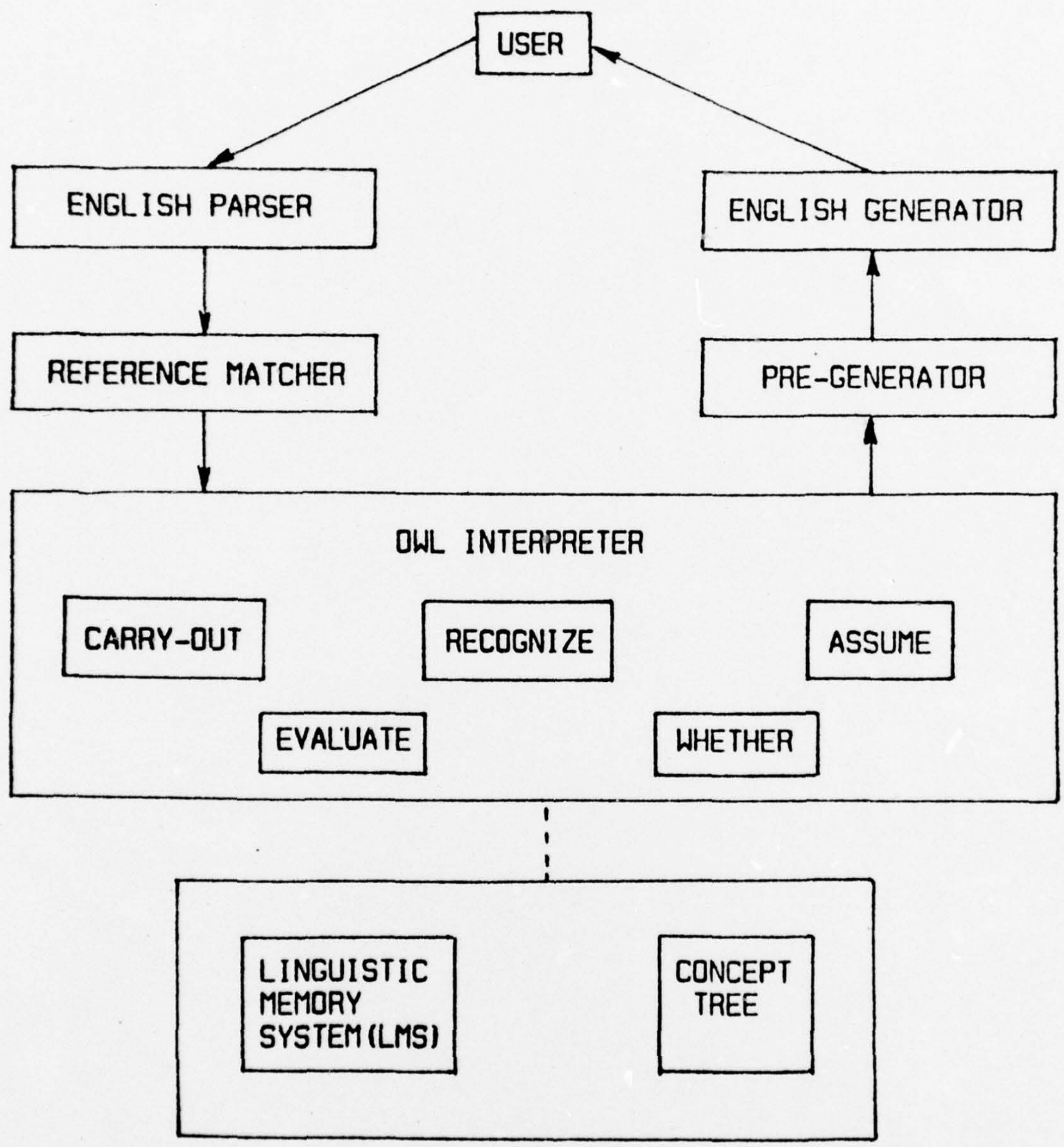

Figure 3. The major components of the OWL-I system.

are found on its *reference list*.

The backbone of the knowledge base is the concept tree. The OWL-I concept tree, which was constructed by William Martin, contains concepts for the words of Basic English (Ogden [12]) plus other concepts of general applicability, among them a set of semantic cases. The concept tree is used by every module in the system, although much of the original organizational impetus was the attempt to reflect regularities perceived in English usage. Each expert system embedded in OWL would bring its own set of concepts to add as specializations of the already-existing tree, and this augmented tree would then be used in both natural language processing and reasoning operations. The concept tree is one (but not the only) place where the analysis of natural language organization is applied to the problem of organizing expert knowledge. Individual concepts will be explained as they come up in the course of the discussion.

Note that the OWL system has opted for a tree rather than a more general hierarchy. This reflects a simplification for purposes of computational efficiency, but it does not constitute a restriction on computational power. While each concept is assumed to have only one *primary* superclass, other class memberships can be entered on the concept's reference list. For example, the concept representing an individual person would be a specialization of the concept HUMAN (although not necessarily a branch), but it might have on its reference list the various roles and properties of the individual, e.g., ADULT, LISP-PROGRAMMER, VEGETARIAN, (AGENT (MOVE BLOCK-A)), etc.

This brings us to the OWL-I Interpreter. The Interpreter executes methods, for example the OWL-I representation of the ask-and-answer dialogue method discussed in Section

3.1. It is the job of the Interpreter module Carry-out to go through the steps of a method, first evaluating them with respect to the current environment. It then matches these evaluated calls against the available methods to find appropriate subprocedures. Restrictions on what can fill each input case associated with a method are used in this matching process. The fact that method selection is a matching process means that it is sensitive to additions to the method library, even though no change is made to the call. The use of the evaluated call in matching allows the choice of method to be routinely dependent on the current operating environment, which introduces considerable flexibility. It also means that one cannot predict *a priori* which method will be selected for a call. This fact has significant implications for recognition, as we will see in Section 7.1. Note also that in the OWL-I system no attempt is made to simulate distributed control, e.g., in the use of demons. Execution of methods is highly centralized, guided by a single control loop in the Interpreter.

A record of the execution process is kept in the *event tree*, which is also used by the Interpreter in making control decisions. The events on the tree correspond to the substeps of the methods executed. Past events are not removed from the tree, so that they are available for inspection, question answering, resumption (in the case of uncompleted events), etc. It is important not to confuse the event tree with the concept tree. The former is built by the Interpreter as a record of methods executed in the course of a console session, while the latter is a part of the knowledge base, embodying the first cut at organizing all of the concepts known to the system, not only the events. In this regard, the event tree can be thought of as intermediate term memory, used to record the current problem solving session and organized chronologically. The concept tree can be thought of as long term memory and is not organized chronologically.

Several Interpreter modules shown in Figure 3 come into play in the course of carrying out a method. The module Evaluate takes OWL-I forms and returns instantiations with respect to some environment, for example when a call is evaluated before the search for a method to carry it out. The module Whether takes predicates and tells whether or not they hold in the current operating environment. To do this, it uses a combination of built-in strategies and user-supplied procedures.

Two other Interpreter modules, Recognize and Assume, were added specifically to handle dialogue. Recognize develops and maintains expectations, using these to fit user inputs into the ongoing dialogue. This recognition process will be discussed in detail in the sections that follow. The Assume module handles method steps that are carried out by the user but which have no corresponding input to the console, e.g., reasoning steps. There are a number of complex and interesting issues that surround assumptions about the knowledge and mental processes of others, especially those issues surrounding level of detail. The approach taken in the system module Assume is that routine assumption mode processing should be minimal; if, however, failures occur, it may then be necessary to go back and make assumptions in more detail. For example, in order to point out an error in reasoning it might be necessary to have fairly detailed assumptions about the reasoning process involved. The implementation reflects the routine part of this approach, but right now detailed assumptions are implemented in only a very limited way. In routine situations, the Interpreter notes an assumption mode call on the event tree by putting it on the subevent list of the event of which it is a subcall, without making a separate event or doing any method selection. This gives the Interpreter an unbroken record of the paths taken through the methods that have been executed. If later on

it becomes necessary to expand out an assumption, then the call can be used and the expansion can be done, primarily using the procedures in Carry-out.

Two other major modules shown in Figure 3 are the parser and the generator. Since the dialogue routines are not currently interfaced to these modules, they will not be discussed in detail in this paper. The parser and generator do, however, bring up an important distinction between *interpreter level* and *surface semantic* OWL-I representations. As the name implies, interpreter level representation is used by the Interpreter and is the stuff of which methods are made. Surface semantic representation is output by the parser and is also input by the generator. The major difference between the two is that interpreter level representation has undergone more canonicalization than its surface semantic counterpart. In general, where a surface semantic representation will look very much like its surface English counterpart, an interpreter level version of the same utterance will have referents substituted for referring expressions and will have undergone more lexical standardization.

This distinction means that the output of the parser and the input to the generator are not at the same level of representation as that used by the Interpreter. Two intermediate modules are necessary to provide the translation between them: the reference matcher and the pre-generator. The reference matcher takes the surface semantic representation output by the parser and looks for corresponding interpreter level referents, both substantives and events. The pre-generator goes in the other direction, taking interpreter level concepts and finding descriptions for them and ways to express them so that the user will be able to identify the sense intended. A reference matcher is currently implemented, but a pre-generator is not; the implementation, then, currently inputs a surface semantic-type representation and outputs

interpreter level representation. We have been careful to include information necessary for generation at the interpreter level, and the input representation is close to the output of the prototype parser, so that we would not anticipate an actual English interface to cause major changes to the current design.

This accounts for the major OWL-I modules, and we are now ready to consider recognition mode.

5. An Outline of Recognition Strategy

Given an English language input, Susie Software must relate it to the ongoing dialogue; that is, Susie must find and instantiate an appropriate interpreter level representation, since this level of representation is the one used to model the structure of dialogue. Within our framework, then, the transition from English to interpreter level OWL-I constitutes the recognition process. In this section we will outline the strategy used for recognition, starting with a closer examination of the problem.

The basic problem for recognition is the overabundance of alternatives. In the Susie Software system recognition is divided into three subprocesses: parsing, reference matching, and expectation management. The parser may produce more than one surface semantic representation for an input, and the results of a parse may contain referring expressions that match more than one referent. In addition, there are several degrees of flexibility that enrich the set of possible structural expectations. The mixed-initiative environment moves Susie Software in the direction of normal conversation, since, if either participant may change the flow of control, the other participant will have less than complete knowledge about what will happen at any given point. In addition, activities in the Susie environment are not rigidly ordered and disposed of. A new activity may be begun before the old one has been completed, and an activity may be reopened after it has been assumed to be finished. Furthermore, the kinds of exchanges that may occur make it harder to find the boundaries between activities. Giving a description for example, is open-ended in a way that a multiple choice answer would not be. Finally, as we have seen, discussion may occur on more than one level. We not only

have utterances that relate to a task directly, but also utterances that report failure conditions in the ongoing task and metadiscussion, that is, utterances that explicitly alter or clarify the flow of activity.

Given these degrees of freedom, it is clear that processing of an input must be carefully controlled. The parsing strategy developed by Szolovits and Martin [10] intentionally limits the extent of processing. For example, the concepts in the surface semantic representations that are output are chosen to minimize the number of decisions that must be made by the parser. In particular, the parser does not attempt to make distinctions that are not needed to complete the parse. This philosophy is similar to the approach taken by Marcus [9], but it goes beyond it in the extent to which decisions are delayed.

To illustrate the decision-delaying nature of the surface semantic representation, we can consider the different ways to say that one understands some information. One informal way is to say, "I get it." Now, "I get it" in isolation is ambiguous (e.g., Q. "Do you know anyone who gets this journal?" A. "I get it."). A transformation within the parser would have to expand GET into its alternatives, say RECEIVE and (GET IDEA), which is not in the spirit of a decision-delaying surface representation. It therefore seems best to use GET in the surface semantic representation, then depend on interpreter level semantic structures to make further distinctions.

In the reference matcher, too, care must be taken to keep processing under control. Recall that the reference matching process starts with the output of the parser which contains, among other things, concepts corresponding to pronouns and definite descriptions. The reference matching process relates the surface semantic representation to an interpreter level one,

resolving references along the way. The basic philosophy for reference matching has been to exploit both the structure built up on the event tree and the structural expectations (especially the current set of possible standard path successor steps). The implementation currently matches referents present explicitly in the structural expectations, but it does not yet handle referents found elsewhere in the dialogue or referents that are part of general knowledge, independent of the dialogue. We would expect the event tree to be useful in structuring the search for those referents not given explicitly in the expectations (see [5] for such an approach). Whatever the case, for all types of reference matching the process would be driven by structural expectations; whether particular referents are present explicitly or not, the Interpreter will always be matching a user input against *some* structural expectation. Thus, while parsing happens in an identifiably separate pass, reference resolution occurs as needed within the general process of matching surface semantic representations against interpreter level forms.

Having looked at the parsing and reference finding strategies, we can now outline the way that structural expectations are managed by the system. Recall that distinctions among basic utterance types were made in Section 3.3. From special patterns that will be described and from the dialogue methods, the system can derive a set of structural expectations at any given point in the dialogue. The question is, what should the system do with these expectations?

The first issue for expectation matching is the choice between a try-all-possibilities and a stop-on-success strategy. Our original strategy for handling the different recognition mode alternatives was to try all possibilities, and then apply a decision procedure if more than one match was found. Without parallel processing, this approach appears to be infeasible, and we hypothesize that a stop-on-success scheme will be sufficient as long as the match attempts are

ordered carefully. The change in strategy is worth discussing, because it points up some important aspects of the recognition process.

First, in a try-all-possibilities environment, the burden is placed on disambiguation, while in a stop-on-success environment it is placed on ordering. When trying all possibilities, if matching leaves ambiguities then there are two main sources of information: heuristics and the conversational partner. The goal of a stop-on-success scheme, then, should be to incorporate these information sources. If this can be done (at least a large part of the time), then the stop-on-success scheme can perform as well as trying all possibilities and save time in the process.

Looking first at information from the conversational partner, for a disambiguation process the standard mechanism would be to ask for clarification. A stop-on-success scheme, on the other hand, would depend on the partner's ability to catch incorrect interpretations from the responses given. For our constrained environment it appears that it will be possible to frame responses in such a way that the user will know whether or not his intentions were interpreted correctly. If a misinterpretation does occur, the user's next utterance will be something on the order of "That's not what I meant." A general failure method, or a small set of them, could be used to handle this situation.

The second information source is heuristics, and we can give two examples here. The first is a redundancy heuristic: a speaker should not (and therefore usually will not) interrupt a conversational exchange to initiate an essentially identical exchange. This redundancy heuristic is very much like one for entities in a description: if there is no information to the contrary, similar definite descriptions can usually be assumed to have identical referents. A second heuristic can be called the "inertia" heuristic: all else being equal, a context will tend to persist.

Now, in a try-all-possibilities scheme, heuristics of this sort would be built into the disambiguation routines. On the other hand, in a stop-on-success scheme the heuristics would be reflected in the ordering rules chosen. For example, both heuristics given can be incorporated into an ordering scheme easily enough by requiring that standard and recovery path expectations be checked before initiators. This insures that the right match will be found first most of the time.

Since ordering choices are crucial to a stop-on-success scheme, the last topic for this section will be a motivation of the choices made in the system. The following list sums up the ordering of recognition possibilities used in the current implementation if no failure discussion is underway:

(1) Metadiscussion

(2) Standard path successor steps

(3) Recovery path lead-ins

(4) Initiators

(5) General failure method lead-ins

Once a recovery path or general failure lead-in has been processed (either recognized or generated) then the relevant successor steps become expectations and are checked second, in place of standard path successor steps.

Metadiscussion is checked first, since this class seems to be constrained enough that a relatively small number of patterns need be matched. The alternatives are also generally well marked, so that mismatches will tend to be detected rapidly. Next come standard path successor

steps and recovery lead-ins. These are the expectations that vary most as the dialogue progresses. Given the two heuristics above, we want these two classes checked before initiators. Since there are many more ways that things can go wrong than right, the standard path successor steps are checked first. It may be desirable, however, to try interleaving recovery path lead-ins with the standard path successors, so that a standard path successor would be checked and then any recovery paths related to it before the next standard path successor was tried.

We are left with initiators and lead-ins to general recovery methods. The order given is probably the desirable one, although general recovery methods were not used for the sample dialogue, so our experience is limited. It appears that general recovery lead-ins will tend to be difficult to detect, so it is reasonable to leave these for last. Note also that if no tasks are currently underway then this ordering permits initiator patterns to be tried right after metadiscussion patterns, which is what we would want.

This winds up the discussion of ordering. Throughout, the basic recognition strategy has been chosen to keep processing under control. The parser delays as many decisions as possible, the reference matcher takes advantage of expectations developed from the dialogue methods, and match attempts are carefully ordered. Another very important element of the recognition strategy is the mixed matching scheme that has been developed. This scheme is based on the distinctions between the basic utterance types that have been presented, and it is the topic of the next sections.

A final note on the discussion that follows: I will make the simplifying assumption that only one surface representation has been produced for an input. Where more than one surface representation is output by the parser, attempts would be made to match each of them against the relevant structural expectations.

6. Recognizing When a New Task is Initiated

6.1. The Problem

At any time after the initial greetings, the user is permitted to type an initiator, that is, to start up a new top level task, either asking a question or requesting a program. The basic problem for initiator processing, therefore, is to recognize when one has occurred. To do this, Susie takes the user's utterance and tries to form a chain of event nodes connecting the top of the event tree (or, more precisely, an event corresponding to the method step that satisfies user task requests) and the speech act event that corresponds to the user's utterance. If such a path can be built, then the user's utterance is assumed to be an initiator. (The most likely competing possibilities will, however, already have been tried; see Section 5.) This path-building process occurs in an environment of incomplete knowledge, since in most cases the initiator will be the only indication of the user's goals and since many surface forms of initiators are ambiguous.

For an example of initiator processing in recognition mode, assume that the only currently open method is to have a console session and the parser gets the input "Can you pick up the block?". The task of the Interpreter would be to find a speech act step (in the current implementation, ASK, TELL, or COMMAND-REQUEST) that could have produced this utterance, as well as a chain of events that connect the current environment -- the participate-in-console-session activity -- with the speech act step chosen. Figure 4 gives a representation of one such event chain. The purpose of filling in intermediate nodes on the event tree is to allow processing to continue normally after a recognition step. In the example in Figure 4, once recognition is completed, then the next step in the ASK-AND-ANSWER method can be carried

participate in a console session

↓

get to know the answer to a question

↓

ask-and-answer whether Susie can pick up Block-A

↓

understand an ask activity for whether Susie can pick up Block-A

Figure 4. A representation of an event tree path.

out by Susie. The intermediate events, then, not only specify the context for a recognition step but also allow successor steps to be accessed in a routine manner. Here, it is assumed that *the block* in the input refers to Block-A in the current context. There are two conditions that make it difficult to fill in these intermediate events: first, the existence of multiple surface forms and, second, the presence of ambiguity. We consider each of these in turn.

Starting with multiple surface forms, note that in Figure 4 the surface semantic representation is not present as an event on the event tree (although it can be associated with the "understand" event as a property). The interpreter level version of the question, embedded in the event representations, will look something like the surface semantic representation, but it will not be identical. For other utterance forms, the divergence between surface semantic and interpreter level representations will be even greater. Consider, for example, some of the different ways to ask someone to write a program:

(1) I request that you write a program to manipulate blocks for me.
(2) Write a program to manipulate blocks for me.
(3) Would (will) you write a program to manipulate blocks for me?
(4) Could (can) you write a program to manipulate blocks for me?
(5) I want (need) you to write a program to manipulate blocks for me.

(6) I want (need) a program to manipulate blocks written.
(7) I want (need) a program to manipulate blocks.
(8) I would like you to write a program to manipulate blocks for me.
(9) I would like a program to manipulate blocks written.
(10) I would like a program to manipulate blocks.
(11) I request that you give me a program to manipulate blocks.
(12) Give me a program to manipulate blocks.
(13) Would (will) you give me a program to manipulate blocks?
(14) Could (can) you give me a program to manipulate blocks?
(15) I want (need) you to give me a program to manipulate blocks.
(16) I would like you to give me a program to manipulate blocks.
(17) Write me a program, would (will) you?
(18) Write me a program, could (can) you?
(19) Give me a program, would (will) you?
(20) Give me a program, could (can) you?

This is not necessarily a complete list, but it does give an idea of the number of request forms that might come up. These examples would have very different surface semantic representations, but they all would match the same interpreter level COMMAND-REQUEST step. Utterances (3) to (20) are generally called *indirect speech acts*, since the surface form does not correspond directly with the intended speech act. Considering this list, it is clear that the first problem for initiator recognition is the range of forms that an initiator may take.

The problem of relating utterances such as (1) to (20) to their intended speech acts has received a fair amount of attention in the linguistics literature. One approach is that taken by Gordon and Lakoff [6]. Concentrating primarily on requests, Gordon and Lakoff propose a set of four sincerity conditions and then give a single powerful rule to account for the different ways that a request can be framed. There is some question, however, whether this rule is *too* powerful, admitting utterances that are not legitimate requests. Sadock [13] responds to Gordon and Lakoff by criticizing the approaches that try to account for the variety of ways to frame a

speech act by using sincerity conditions and general rules. He distinguishes between the case where an utterance has a speech act as its meaning and the case where the utterance means one speech act but entails another. An example of the first case would be the utterance of "It's cold in here" to convey information, while an example of the second case would be the use of this same sentence to convey a request for someone else to close a window.

With respect to the handling of the surface realizations of a speech act, the Susie Software implementation bears more similarity to the ideas of Sadock than to those of Gordon and Lakoff. Utterance forms corresponding to sentences (1) to (20), for example, are associated with an interpreter level COMMAND-REQUEST representation (or, in the case of ambiguous forms, a procedure which returns COMMAND-REQUEST as one of its values). The way that this association is done is discussed in the next subsection; the association itself can be considered to assign meaning for these forms. Entailed speech acts, on the other hand, would require another level of association, probably between the two speech acts themselves, possibly taking the surface form into account secondarily. Note that entailed speech acts are not now handled by the implementation, although we do not forsee significant problems in adding them.

Having outlined the problem of multiple surface forms, we can turn now to the second problem for initiator recognition, ambiguity. The utterance "Can you pick up the block?" could be either a question, as it was interpreted above, or a request for action. Many surface forms can be ambiguous in this way. In fact, the use of related questions, commands, and statements to signal a speech act all but guarantees ambiguity, since the signals can also be interpreted literally. This ambiguity of forms has social utility since speech acts can be attempted and, if

resistance is met, the speaker can fall back on a less adventurous alternative interpretation (a question instead of a command, for example). This dimension is not, however, particularly relevant in our environment, so that ambiguity is viewed here essentially as a problem.

To recognize an initiator, then, it is necessary to construct an appropriate event tree path. The primary difficulties that confront the system in this effort are the existence of a variety of indirect speech acts and the inherent ambiguity of particular forms.

6.2. Initiator Keys

In the Susie Software environment, context is not particularly useful in predicting the nature of new tasks, so the initiator recognition scheme is essentially bottom-up; that is, the event corresponding to the user's utterance is determined first, then superior events are determined until the entire path has been accounted for. To facilitate this process, special structures called *keys* are used; the main purpose of keys is to provide patterns for matching initiators and constructing the appropriate context. In the current implementation, keys are represented using a special type of OWL-I concept called a *relation*. The important attribute of relations is that they can have an associated *value*, which is assigned using a left arrow. The relations used for initiators are specializations of the concept INITIATOR-KEY; later on we will see another kind of key used for metadiscussion.

Initiator keys come in two varieties: terminal and non-terminal. Terminal keys are used to match a surface semantic representation (i.e. parser output) against the actual speech act step in a dialogue method (e.g., the "understand" event in Figure 4), while non-terminal keys are

used to fill in the intervening events between the speech act step and the console session event on the event tree (e.g., the get-answer and ask-and-answer events in Figure 4). Note that although the Susie environment is restricted to two top level activities (program writing and question answering), the patterns in the keys are more general than this; they match requests for information of any sort and requests for any sort of activity with the user as beneficiary. So, although the keys are special-purpose patterns, their usefulness is not restricted to the Susie Software environment.

Terminal keys take the following form:

(<key type> <a subsurface level form>) <-- <a method call>

The first two parts, enclosed in parentheses, are represented by a single OWL-I relation, with the third part represented by another concept that becomes the value of that relation. The relationship between the parts will be discussed in more detail below, but first we must clarify what is meant by "subsurface level." A subsurface level form is, as the name implies, at a level of representation intermediate between surface semantic and interpreter level. Subsurface representations have only been used in the special matching patterns and for records kept in the recognition process. (They would also be used as input to the pre-generator, if this were operational.) There is not space here to go into the difference between subsurface level and the other levels of representation in detail, but, basically, subsurface representations differ from surface semantic level ones in that they may contain variables that may be bound as part of a matching process. They differ from interpreter level forms in that they mirror the surface form of an utterance rather than its underlying speech act. For example, "I want to know the

color of Block-A" would match a subsurface form that is a specialization of the concept (SAY DECLARATIVE), reflecting the declarative nature of the surface form, but the underlying speech act would often be interrogative, a request for information. Thus, the interpreter level form that matched this utterance would be a specialization of ASK. One identifying characteristic of subsurface forms is the fact that they are represented as specializations of SAY, while interpreter level forms of speech acts are currently represented as specializations of TELL, ASK, and COMMAND-REQUEST.

Returning to the basic form of terminal keys, the system matches the output of the parser against the keys' subsurface level forms. If a match is found, then the value of the matching key, the method call, is retrieved. This method call is either an interpreter level representation of the underlying speech act or a call to a special OWL-I disambiguation method. If there is no ambiguity, then the key relation's value can be used for the next stage of the match against non-terminal keys, which will be discussed. If, on the other hand, a call to a disambiguation method is found, then the procedure is executed to return a speech act representation, which can then be used for the next match stage. (More on disambiguation methods is given below.) Note that the structure of keys shown here is the form produced by the programmer. When the keys are loaded into the system, the LMS Reader adds them to the concept tree, automatically creating a key subtree which can be used in matching. (See Hawkinson [7].)

Once the event for the speech act has been found, non-terminal keys are used to fill in the higher events on the path. The basic form for a non-terminal key is as follows:

```
(<key type> <a method call>)
    <-- <either: a call to the method that contains this step
                 on the initial path of a top level activity
            or: a call to a disambiguation method>
```

Both the first method call and the value of a non-terminal key are interpreter level representations. The system starts with the result of the terminal key match (or the result of processing the associated disambiguation method) and matches this against the method calls in non-terminal key relations. Just as for terminal keys, when a match is found the value is retrieved, and it can be used either to form a higher event on the event tree path or to call a special disambiguation method to choose the correct higher event. Matches against non-terminal keys are repeated until the value retrieved can form a subevent of the general console session event, completing the path.

We can now return to the topic of disambiguation. In the last subsection we observed that indirect speech acts are by nature ambiguous, since they can be used either for the underlying speech act or for the speech act conveyed explicitly in the surface form. A system therefore needs a mechanism to handle this routine sort of ambiguity. For initiators we have used special OWL-I methods, all of which belong to the class DISAMBIGUATE. The fact that disambiguation routines are coded in OWL-I means that they can be examined and explained by the Interpreter. This is part of the general policy that takes as many choices as possible out of black boxes, allowing inspection and evaluation. It also means that the reasoning processes needed to do disambiguation can be done by the Interpreter module Whether. The disambiguation methods are flexible, with several different strategies available for use, among them a last-resort request for further clarification by the user.

This almost concludes the discussion of initiator recognition. Remember that initiator keys are used only for initiators, which are lead-ins to the two top level tasks. Thus, not all procedures and utterance types will have associated initiator keys -- just those that lie on the

first event tree branch of a top level task. Although the set of top level goals could be extended, there would still remain substeps and procedures without associated keys. Thus, one justification for using a special matching structure rather than a more general search procedure is that initiators are a subset of the possible lead-ins to dialogue methods. Using this special structure, the Interpreter will not try to construct paths that are known *a priori* to lead to deadends. Similarly, a general bottom-up search mechanism that tried to construct a path from initiator to top level activity would be slowed down considerably by the fact that initiators are not necessarily produced by the first step in a method. (Recall that they are produced by the first step executed in *recognition mode* when Susie is not the agent of the method.) Given the way that OWL-I method steps are linked, the fact that a path could not be reliably constructed from first steps would make a general search mechanism without special structures inefficient.

Initiator keys, then, perform three functions. First, they provide an efficient way to find the event tree path between an utterance and the console session activity. Second, and related to this first point, keys provide calls to disambiguation procedures at appropriate points. Third, keys contain information about the indirect speech act forms taken by initiators.

6.3. When Not to Match Against Initiator Keys

The last topic for this section is whether initiator matches must always be attempted. Are there types of inputs for which matches against initiator keys are clearly ruled out? First, in typed dialogue it appears that for two sorts of input the initiator possibilities need not be tried at all:

(1) Sentence fragments, including placeholders such as *O.K.*

(2) Complete sentences beginning with *yes* and *no*[5]

For these two categories of user input, there is no need to match against initiator keys.

5. The only exception to (2) appears to be when *yes* is used to establish a helping or command relationship (e.g., "Yes, I'd like to know when the 5:15 train gets in to Portland."). This seems to be in answer to an assumed "Can I help you?" This special case should be easy to screen, since it is always found at the beginning of the conversation after the hello-ing is done.

7. Fitting a User Input to Open Tasks

Susie Software uses the dialogue methods to generate structural expectations that are dependent on the course of the console session to date. In this section we consider the situation in which execution of one or more methods is already in progress and the next step on a standard path is to be executed in recognition mode. To perform the appropriate match, the Interpreter first needs the ability to detect possible next steps, and, second, it needs to do the actual matching as efficiently as possible. These two topics are discussed in the subsections that follow.

7.1. Expectations

For a given step in a method, how many possible standard path successor steps are we talking about? There is, of course, the possibility of branching within a method, using the OWL-I conditional IF-THEN. In addition, an IF-THEN that is not inside the scope of an OR effectively makes the consequent an optional step, so that the Interpreter must be ready to recognize either the optional step or its successor. Another complication arises from the fact that the ends of some steps are not always clearly marked (e.g., in giving a description). The Interpreter must be ready, then, to detect the completion of a step by matching against steps that continue methods higher on the event tree.

This seems to exhaust the possibilities, since there are some sorts of situations that probably do not come up. In the specifications for Susie Software, we admit the possibility of

carrying out two activities (e.g., writing two programs) by alternating steps. To switch from one activity to another and back, however, it appears that the rules of dialogue require the use of metadiscussion. The user would have to specify which task a particular utterance applied to. Similarly, to re-open a previously completed or suspended activity, the user would have to state his intentions explicitly. Neither of these situations, then, would have to be handled by the normal standard path expectations mechanism. We are therefore left with a relatively small number of possible standard path successor steps; ten possibilities probably borders on pathological for this environment.

With such a small number of possible standard path successors, how could the system possibly run into trouble? Unfortunately, there is not far to look. Two difficulties come up here. First, as with initiators, there may be a variety of indirect speech act forms used to convey a single interpreter level standard path successor. Second, the OWL-I Interpreter's method selection mechanism presents special problems for recognition. The first issue is considered in the next subsection, while the second is discussed below.

Recall that OWL-I method selection involves a match on the call evaluated in the current operating environment so that it is not in general possible to predict *a priori* which method will be selected to carry out a particular call. Compared to other processes in the system, the method selection process is a relatively expensive one. The problem for recognition is that a standard path successor step will not necessarily be a simple speech act step (currently, TELL, ASK, or COMMAND-REQUEST) but may instead be a call to a general dialogue method which effects, at some level of embedding, the simple speech act. We will call standard path successor steps that are not themselves simple speech acts *non-terminals*. For example, when Susie asks

the user for a description of the input and output conditions of the desired program, she executes a COMMAND-REQUEST step. The recognition mode step that follows is not, however, a TELL but instead a non-terminal call to STATE-AND-ACKNOWLEDGE which itself contains a call to TELL. In general, it may be necessary to go through several layers of calls and procedure selection processes before the actual speech act step is found.

The dialogue system has gone through several different implementation phases in an attempt to deal with this problem. Briefly, the fact that method selection depends on matching in the current operating environment makes it difficult to compile methods into simple speech acts so that non-terminal calls would effectively be eliminated. It is also not clear that compilation presents a long term solution, because intermediate event tree nodes may convey important information for the purposes of reference resolution. A second alternative is a straight top-down expansion of the non-terminal call at recognition time; however, the fact that method selection is a relatively expensive operation makes this unattractive, since the worst case occurs when the utterance fails to match the expectation at all. Finally, although the core methods do not tend to take advantage of the full power of the OWL-I method selector, this does not help us in recognition, since these problems come up for any OWL-I method that has, at some level, a simple speech act as its lead-in.

The non-terminal recognition scheme currently implemented is a mixture of top-down and bottom-up matching that relies on the programmer's ability to predict the set of dialogue methods that will be possible matches for the non-terminal call, and then continue this prediction process on substeps until speech act steps are found. The set of possible lead-ins for a method is specified using the OWL-I relation LEAD-IN, which takes the form:

(LEAD-IN <method-1>)

<--

<the speech act steps that are lead-ins to method-1>

Note that method-1 above need not have the speech act steps as immediate substeps; this situation, in fact, is one in which the special LEAD-IN structure is particularly useful. Given a LEAD-IN relation, the input utterance is matched against the values in turn. If no matches are found, then this standard path successor step can be abandoned, thereby minimizing the time taken for many of the non-matches.

If the input matches one of the values of the LEAD-IN relation, then the Interpreter can construct an event for the basic speech act step and an event for its containing method, which is recoverable from any variables in the matching LEAD-IN value, due to the way that OWL-I variables are constructed (see the appendix). The construction of the events for the basic speech act step and the containing activity is the bottom-up part of the process, and once this is done an attempt is made to attach the new events to the event tree. If, however, intervening events are still needed, then the event path is constructed top-down until the bottom-up event fragment can be attached. Note that the new path constructed is not attached to the event tree until it is completed, in case this path turns out to be incorrect and must be discarded. In the methods for the sample dialogue and in other examples considered, calling depth for non-terminals and substep fan-out was limited enough to keep the top-down expansion phase under control. It is not clear at this time whether this scheme will continue to be sufficient as the library of methods grows.

One other problem that is always with us is ambiguity. For standard path successors, at

least within the framework of the current matching scheme, it does not appear that we will find disambiguation mechanisms to be as important as they are elsewhere in the system. With a stop-on-success matching scheme, ambiguities among possible standard path successors are not detected directly; where one occurs, the system must rely on the user to detect any errors made. In a try-all-possibilities scheme, on the other hand, disambiguation plays a more prominent role, in that more than one standard path successor could match an utterance. But, even in this case, it appears that our domain is structured in such a way that few ambiguities will occur between standard path possibilities and most will occur across types (e.g., an utterance might be either an initiator or a standard path successor). To the extent that standard path ambiguity is detected by the system, the remedy would seem to be the use of a few general purpose disambiguation procedures (such as asking the user for more information) rather than depending on special purpose methods. This is because it will not in general be possible to predict the sorts of ambiguities that will occur between standard path successor steps in the same way that it is for, say, initiators.

Given this general discussion of expectation management for standard path successors, we can now consider the handling of indirect speech acts.

7.2. WAY Evaluators

Just as for initiators, there will tend to be several different ways to phrase a next-step utterance. To access these different utterance forms, we can use a method type called an *evaluator*; the similarity between this name and the Evaluate module is intended. An evaluator

is an OWL-I method with a header that is a relation and a standard path that gives criteria for choosing between possible values. The evaluators that are used here are WAY evaluators, which give alternate ways to convey the same interpreter level message. The use of WAY evaluators will be explained for recognition, but they are also intended to be used for generation.

Figure 5 gives an English translation of a WAY evaluator. This particular example is a relatively simple one, but it is useful for illustration. It is intended to apply to situations where a statement has just been made and the acknowledgment given is meant to convey that the statement has been understood. Among the ways to phrase the acknowledgment are, "O.K.," "I understand," "I see," and "I get it." There are no doubt more possibilities and the conditions for each could be tightened up, but this example is enough to sketch out the approach that has been taken. What the WAY evaluator offers is a link between certain interpreter level representations (e.g., the representation for acknowledging a statement) and a set of subsurface level forms (e.g., the representations for the ways to frame the acknowledgement). In the actual OWL-I method, the alternative forms would be assigned to the output case PRINCIPAL-RESULT, so that a given choice would be considered the result of the method. In its primary use the WAY evaluator would be run as part of the generation process, to select a basic subsurface semantic form for a speech act call in the current environment. The phrase *in the current environment* is important here because it justifies the use of a procedural form; otherwise, a simple list of the possibilities would be sufficient. The subsurface form supplied by the WAY evaluator would be passed in to the pre-generator to start the generation process.

For recognition mode matching, the Interpreter will be starting with an ASK, TELL, or COMMAND-REQUEST step. Using this step, the recognition module constructs an OWL-I

find a way to acknowledge a statement
  object: a variable for the ways to acknowledge a statement
  agent: a person or computer system

  method:

   either
    (1) If the context is informal and it was a routine process
      to integrate the statement in with existing knowledge,
      say "O.K."

    (2) If the context is informal but the integration process
      was not routine, say "I get it."

    (3) If the context is formal and the integration process
      was quite difficult, then say "I see."

    (4) If the context was formal and the integration process
      was not routine, then say "I understand."

Figure 5. An English representation of a method for choosing a way to acknowledge a statement.

method call, then uses the Interpreter's method selection routine to choose an appropriate WAY evaluator. The PRINCIPAL-RESULT alternatives of the evaluator found can then be used to match against the surface representation of the user's input. Either one subsurface level alternative will be found to match or, if none do, then the successor step can be eliminated as a possibility altogether. Once a match is found, the implications of a user's choice can be derived by an inspection of the path of tests leading to that particular result. While these implications are not particularly important for our purposes right now (since many center on politeness and authority relationships), they are important in human-to-human interaction and might prove useful as we try to fine-tune the system.

To sum up, WAY routines are limited in their uses and simple in their structure, but their style is important. They are special-purpose structures to perform a function that might be done by a general deduction mechanism in other systems. By differentiating among the sorts of inferences that must be made, we can isolate special bits of knowledge and know exactly when they are to be accessed. Finally, note that part of the information present in WAY evaluators is also found in initiator keys. The information is represented in different ways, however, reflecting the hybrid approach taken in successor step matching as compared to the essentially bottom-up approach taken in initiator matching.

8. The Best-Laid Plans: Recognition in Failure Situations

The next topic is what happens when a failure situation either occurs for the user or is detected by him. The user either reports the failure explicitly or the failure is implicit in an utterance, and it is the job of the Interpreter to do the necessary recognition. How can this be done? We first discuss the access of expectations for failure conditions and then look at matching. A more general discussion of failure handling in a conversational environment is given in [1].

In section 3.3, recovery discussion was divided into two types, that represented by recovery paths and that represented by general OWL-I recovery methods. Naturally, access of failure expectations will differ with the type of representation. Recall that autonomous recovery methods are not needed for the sample dialogue, so this area of the design was not emphasized. Existing mechanisms could be used to access and match these recovery methods, but it is not clear that this will be adequate if the number of recovery methods is large. At this time, however, only a small range of failures seem to require the generality of recovery methods, so it seems better to delay any speculation about the sort of access mechanism that would be required until more experience has been accumulated.

This leaves the problem of recognizing recovery path lead-ins. When recovery paths are accessed in carry-out mode, a search up the event tree path is done, using the type and site of the failure to select the appropriate path. Where more than one match occurs, the one highest on the event tree is chosen, allowing context to affect the selection in a modest way. For recognition mode, however, the failure is not available (since it occurred for the user), so it is

necessary to check the lead-in for each recovery path on the event tree path, matching it against the incoming utterance. There may be a fair number of recovery paths to be inspected, but the number will be far smaller than the total set.

The access mechanism is complicated slightly by the existence of assumption mode steps. For example, Susie asks a question. The user then tries to understand the question and finds the answer. The user's next standard path successor step would be to give the answer, and this is Susie's first recognition mode step. If, however, something has gone wrong, then the user's next utterance may be part of a recovery path. The failure could have occurred in the understanding process (e.g., "I don't know what you mean by..."), in the answer finding process, or it could be related to the process of giving the answer. Thus, it is necessary to check recovery paths related to assumption mode steps as well as steps executed in recognition mode. Where more than one possible site of the failure exists, we have been checking the possibilities in the temporal order specified by the methods, since this seems to be as good as any.

One feature of recovery paths that was mentioned in passing above is that they may be associated with higher level methods (those methods that call the method in which the failure occurs); therefore, the particular recovery paths available for a given failure become context dependent. This feature causes no special difficulties for recognition, and in fact it sometimes provides an advantage. If a failure occurs inside a step executed in assumption mode by Susie, then it may be necessary to go back and add an event and subevents for the assumption step to the event tree, in order to get at recovery paths. If, however, a recovery path associated with a higher event is the one used, then this expansion will not be necessary.

The next topic for this section is matching. Basically, recovery path lead-ins can be

handled in a manner similar to standard path successor steps, i.e. by matching on PRINCIPAL-RESULTs of WAY evaluators. The mechanism used for handling non-terminal standard path successor steps was also adopted here. The only difference between standard and recovery path recognition steps seems to lie in the WAY evaluators themselves. Some utterance types seem exclusively failure-related, such as the statement of a lack of information as a way to request it (e.g., "I don't know X.") Moreover, the connotations of some utterances seem to differ according to whether they are used as recovery path lead-ins or in other contexts. These two facts together indicate the need for at least some independent structure for WAY evaluators for recovery paths. Where independent recovery path WAY evaluators entail too much duplication, a combined structure could be used, with properties distinguishing exclusively failure-related usages from others.

Recognition of recovery discussion, then, is a relatively straightforward process. We have concentrated on the use of recovery paths, where access is similar to that for carry-out mode, and matching is similar to that done for standard path successor steps.

9. Metadiscussion

Metadiscussion is used to change the flow of activity in a dialogue or clarify the current flow of activity. In the system, metadiscussion recognition is done using a set of patterns called *metadiscussion keys*. These are similar to initiator keys, except that metadiscussion keys are not a reflection of other structures (i.e. method steps) in the same way that initiator keys are; instead, they themselves are the primary structure.

In the Susie Software environment, three sorts of metadiscussion are possible: either an activity is suspended, a previously suspended or completed activity is reopened, or a description of what is to come next is given. Accordingly, the concepts for metadiscussion keys are specializations of the relation METADISCUSSION-KEY, with this category divided into three subcategories marked by the concepts SUSPEND, RESUME, and INFORM. The basic form of a metadiscussion key is as follows:

(<key type> <a subsurface level representation>) <-- <the corresponding method call>

The matching process for metadiscussion keys is similar to that for terminal initiator keys. (See Section 6.2.) The surface semantic representation output by the parser is matched against the subsurface level forms of the keys until a match is found, and then the associated value is picked up. The value of a metadiscussion key is an interpreter level call to a method, for example a call to the method to resume an activity. Because of the way that keys are represented, a match against the subsurface level form causes the appropriate binding of its variables and at the same time causes the method call in the value to be appropriately

instantiated, so that the normal Carry-out method selection process can be applied to it. Once recognition of the user's utterance is complete, then, the value of the metadiscussion key can be used to guide Susie's response.

So far, metadiscussion processing seems to be quite simple, but there is one complication. The operational definition of metadiscussion used in the system is that it is a sentence or phrase that changes the way other sentences or phrases are interpreted, in particular by changing in some way the set of pattern matches that is tried. Note that, as we are construing it, not all metadiscussion is sentential. Some seems to be phrasal, as in *By the way*, *While we're on the subject*, *Coming back to the last program*, etc. Besides these more or less stock phrases, it appears that general time phrases can be used as metadiscussion. Consider the following example from Deutsch [5]:

A: I have the jaws around the hub. How should I take it off now?

E: Tighten the screw in the center of the puller...that should slide the wheel off the shaft.

A: OK. It's off.

A: A little metal semicircle fell off when I took the wheel off.

Deutsch observes that the phrase *when I took the wheel off* is used to reopen an already completed subtask. In our framework, this phrase would be classified as metadiscussion associated with the reopening of a task to initiate a recovery procedure. Time phrases are not, however, restricted to this usage; for example, "After you pulled the plug, did the water run out?". Within our framework, the difference between the two uses of time phrases is that in the first case, once a subtask has been closed by a successful recognition step ("O.K. It's off."),

associated recovery procedures are no longer directly accessible. The time phrase is therefore a necessary cue to the system to return attention to the event named, making the patterns needed to interpret the rest of the utterance available. In the second example, however, the necessary patterns (in this case, initiator keys) are routinely available. The time phrase is necessary to answer the question, but it is not necessary for finding the correct pattern in the first place.

When metadiscussion forms a complete utterance, the system can match the appropriate metadiscussion key and then carry out the call given in the value. When metadiscussion is phrasal, on the other hand, the system will again be matching metadiscussion keys, but the activities given as values for these keys will have to specify further matching operations in order to complete the processing of the utterance and, in doing so, verify that the phrase was actually acting as metadiscussion.

10. Conclusions

This paper has described a framework for processing mixed-initiative typed dialogue, with special attention to recognition. Recognition is done using a set of dialogue methods written in OWL-I, a set of special recognition patterns, and multiple matching strategies. The dialogue methods and the recognition patterns are used to provide structural expectations, some of which are developed dynamically. The task-oriented nature of the environment means that the structural expectations will be a relatively good source of information, and this in turn allows a good deal of flexibility to be incorporated into the Susie Software system.

The important question for a natural language system of this sort is its extensibility. Can the design presented here be adapted for real-world interaction with an expert system?

First, the dialogue methods include both semantic domain dependent plans (e.g., the program writing one) and domain independent ones, the core methods. If we were to add new tasks to the Susie Software environment the core dialogue methods would continue to be applicable. Moreover, taking Searle's speech acts as a guide, it is probably possible to write a complete set of core dialogue methods. (The number of these would be on the order of a hundred and certainly less than a thousand.) Beyond the accompanying recognition patterns and methods, no new structure would have to be added to the system to accomodate these new core methods, and at one level we could then say that we had a very general system.

There is more to dialogue, however, than domain independent structures, and the question of extensibility is more problematic *within* the domain that has been chosen. In order to provide users with a working system, we would have to span the semantic domain. This

requires more domain dependent dialogue methods, which would be relatively straightforward. Beyond this, it requires special structures and a good deal of built-in knowledge to handle reasoning, reference resolution, and the framing of messages for generation. Crucial to this is also a facility for modelling the knowledge of the user. These issues have not been ignored in system development, but more work, some of it theoretical, would have to be done before we could start to talk about Susie Software as a practical system.

Keeping these reservations in mind, we feel it is fair to say that the Susie Software design constitutes a step in the direction of flexible, reliable, task-oriented dialogue processing. The design incorporates different kinds of knowledge needed to model dialogue, and it accomodates discourse phenomena such as indirect speech acts and intersentential reference. Our experience with the prototype system has been positive, and we feel that the approach taken in Susie Software is a promising one.

References

[1] Brown, G.P. Failure and recovery in natural language dialogue, in progress.

[2] Brown, J.S., Burton, R.R., and Bell, A.G. SOPHIE: A sophisticated instructional environment for teaching electronic troubleshooting (an example of AI in CAI), BBN Report No. 2790 (March 1974), Bolt, Baranek, and Newman, Inc., Cambridge, Mass.

[3] Bruce, B.C. Belief systems and language understanding, BBN Report No. 2973 (January 1975), Bolt, Baranek and Newman, Inc., Cambridge, Mass.

[4] Carbonell, J. *Mixed-Initiative Man-Computer Instructional Dialogues*, Doctoral thesis, Electrical Engineering Dept., M.I.T., Cambridge, Mass. 1970.

[5] Deutsch, B. G. The structure of task oriented dialogs, *Proceedings IEEE Speech Symposium*, Carnegie-Mellon University, Pittsburgh, Pa. (April 1974).

[6] Gordon, D. and Lakoff, G. Conversational postulates, in: Cole and Morgan (Eds.) *Syntax and Semantics*, vol 3., Academic Press, New York, 1975.

[7] Hawkinson, L. The representation of concepts in OWL, *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR (September 1975).

[8] Levin, J.A. and Moore, J.A. Dialogue games: meta-communication structures for natural language interaction, ISI/RR-77-53 (January 1977), USC/Information Sciences Institute, Marina del Rey, Calif.

[9] Marcus, M. Diagnosis as a notion of grammar, *Theoretical Issues in Natural Language Processing*, An Interdisciplinary Workshop in Computational Linguistics, Psychology, Linguistics, and Artificial Intelligence, Cambridge, Mass. (June 1975).

[10] Martin, W. A. A computational approach to modern linguistics, in progress.

[11] Minsky, M. A framework for representing knowledge, in: Winston (Ed.), *Visual Information Processing*, MIT Press, Cambridge, Mass. 1975.

[12] Ogden, C.K. *Basic English*, Harcourt Brace and World, New York, 1968.

[13] Sadock, J. M. *Toward a Linguistic Theory of Speech Acts*, Academic Press, New York, 1974.

[14] Schank, R.C. and Abelson, R.P. Scripts, plans, and knowledge, *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR (September 1975).

[15] Schegloff, E. and Sacks, H. Opening up closings, *Semiotica*, 8 (1973).

[16] Searle, J. R. *Speech Acts*, University Press, Cambridge, 1969.

[17] Searle, J. R. A taxonomy of illocutionary acts, in: Gunderson (Ed.), *Minnesota Studies in the Philosophy of Language*, University of Minnesota Press, Minneapolis (forthcoming).

[18] Szolovits, P., Hawkinson, L., and Martin, W.A. An overview of OWL, a language for knowledge representation, to appear in: *Proceedings of the Workshop on Natural Language for Interaction with Data Bases*, International Institute for Applied Systems Analysis, Laxenburg, Austria.

Appendix

For readers interested in more detail, we include a brief survey of OWL-I notation, along with an example of an OWL-I method.

Recall that there are two fundamental operations in OWL-I, specialization and modification. Specializations of a concept are represented using parentheses, e.g., (NAME FIRST) for "first name," a specialization of NAME. FIRST is called the *specializer* of (NAME FIRST). In OWL-I, identical concept names correspond to identical internal structures, so that two different uses of (NAME FIRST) will have the same internal representation. To represent the fact that a concept modifies another concept, we use square brackets to form a *complex*, e.g., [PAPER OFFICE-SUPPLY]. This says that the concept for paper has the concept for office supply as a modification. Note that OFFICE-SUPPLY is actually a *label* for a concept that might also, for example, be written as (SUPPLY OFFICE). In general, labels are used to increase readability, and they are assigned using an equal sign, e.g., OFFICE-SUPPLY=(SUPPLY OFFICE). A special position on the reference list is reserved for *values* of relational concepts such as EMPLOYER, SUPPLIER, LENGTH, WIDTH, etc. The notation for value assignment is a left arrow; for example,

(EMPLOYER MARY-DOE) <-- UNION-CARBIDE

says that the employer of Mary Doe is Union Carbide. A question mark may be used after a relation to refer to the value, so that the following representations are equivalent:

(EMPLOYER MARY-DOE)?   (VALUE (EMPLOYER MARY-DOE))

The Interpreter can take either of these forms and evaluate them to return the current value of the relation. Mechanisms exist to handle values that change over time and also to handle values that are context or world model dependent.

As an abbreviation for specialization by the first concept (the *subject*) of a complex we use colons. Thus,

```
[BLOCK-A   COLOR: <-- RED]
```

is equivalent to:

```
[BLOCK-A   (COLOR BLOCK-A) <-- RED]
```

Both say that the color of Block-A is red. The number of colons corresponds to the level of embedding of the square brackets, so that on input the expression

```
[((PUT ENTITY) ((ON TOP) ENTITY))
 INSTRUMENT:  <--  [HAND:
                     (PART AGENT:) <-- ::]]
```

would be equivalent to

```
[((PUT ENTITY) ((ON TOP) ENTITY))
 (INSTRUMENT  ((PUT ENTITY) ((ON TOP) ENTITY))) <--
     [(HAND  ((PUT ENTITY) ((ON TOP) ENTITY)))
      (PART (AGENT  ((PUT ENTITY) ((ON TOP) ENTITY)))) <--
          (HAND  ((PUT ENTITY) ((ON TOP) ENTITY)))]]
```

Both structures above express the constraint that the instrument case of the concept ((PUT ENTITY) ((ON TOP) ENTITY)) must be bound to the agent's hand. Specialization by the subject of a complex is used to tie concepts into larger structures. For the OWL-I Interpreter, colons most often indicate that a concept is to be used as a variable.

We are now in a position to look at the actual OWL-I representation for the method to ask and answer a question, Figure 6. The subject of the whole complex (the method header) is ASK-AND-ANSWER; this means that the other concepts that follow will appear on its reference list. From the point of view of LMS, the concepts on the reference list of the ASK-AND-ANSWER header differ by type, but they are basically semantically neutral. For the Interpreter, however, there are important semantic distinctions between them, some of which have already been mentioned. The first of these concepts is PLAN. The ASK-AND-ANSWER concept is thus characterized as a PLAN, which makes it possible for the Interpreter to distinguish it from individual ASK-AND-ANSWER events. Next in Figure 6 are the semantic case specifications; for methods, these come from a set of twenty cases (which may, however, be further specialized). Before discussing the content of the case specifications in detail, it is necessary to clarify the role of colons, numbers, and THE concepts in method notation.

At the LMS level, we have said that colons are used to tie concepts into larger structures. This also serves to distinguish different uses of a concept. For example, the concept AGENT has a unique representation in LMS, so the uses of AGENT in [ASK-AND-ANSWER AGENT <-- ...] and [STATE-AND-ACKNOWLEDGE AGENT <-- ...] would refer to the *same* concept. If, on the other hand, we follow AGENT with a colon in these two complexes then there are two *different* concepts: (AGENT ASK-AND-ANSWER) and (AGENT STATE-AND-ACKNOWLEDGE). If this degree of distinction is not enough, for example if it is necessary to distinguish between two concepts within a complex, then OWL-I notation uses specialization by a number. Thus, HUMAN:1 and HUMAN:2 in ASK-AND-ANSWER

```
[ASK-AND-ANSWER
  PLAN
  OBJECT: <-- [SUMMUM-GENUS:1  NON-HOW-WHY-QUESTION
              (GOAL AGENT:) <-- (KNOW (ANSWER ::))
              ((BE ((INFORMATIONALLY-NONSPECIFIC -SELF)
                                  (FOR CO-AGENT:))) ::)]
  AGENT:   <--  (OR HUMAN:1 VERBALIZER:1)
  CO-AGENT: <--  (OR HUMAN:2 VERBALIZER:2)

 METHOD: <--
  [(ASK OBJECT:)
       AGENT::       <-- AGENT:
       DESTINATION:: <--  CO-AGENT:],

  (BECOME  ((BE (SPECIFIC -SELF)) '(OBJECT: (FOR CO-AGENT:)))),

  [((FIND MENTAL)
          ([SUMMUM-GENUS:3   (ANSWER OBJECT:) <--  :::]  SOME))
      AGENT::        <--  CO-AGENT:
      BENEFICIARY:: <--  AGENT:],

   [(STATE-AND-ACKNOWLEDGE  (SUMMUM-GENUS:3 THE))
       AGENT::   <--  CO-AGENT:
       CO-AGENT:: <--  AGENT:]

((RECOVERY-PATH STIPULATION) ((FIND MENTAL) SUMMUM-GENUS:3)):
     <--
     [(TELL (AND (SUMMUM-GENUS:3 THE)
              (STIPULATION (PRINCIPAL-RESULT
                  ((FIND MENTAL) (SUMMUM-GENUS:3 THE)))?)?))
      AGENT::        <--  CO-AGENT:
      DESTINATION:: <--   AGENT:],

    [(TELL [SUMMUM-GENUS:4 AFFIRMATIVE])
      AGENT::        <--  CO-AGENT:
      DESTINATION:: <--  AGENT:]

((RECOVERY-PATH (KNOW NOT)) ((FIND MENTAL) SUMMUM-GENUS:3)):
     <--
     [(TELL (AND (BE SORRY)
              ((KNOW NOT) (WHAT (ANSWER OBJECT:)))))
       AGENT::        <--  CO-AGENT:
       DESTINATION:: <--  AGENT:]]
```

Figure 6. An OWL-I representation for the process of asking a question and getting an answer.

expand to ((HUMAN ASK-AND-ANSWER) 1) and ((HUMAN ASK-AND-ANSWER) 2), two distinct concepts. Finally, we can go further and specialize by THE. This last distinguisher will be discussed in more detail below.

Returning now to the contents of the semantic case specifications, in order for the ASK-AND-ANSWER method to run the agent must be bound to a kind of person or computer system, and so must the co-agent. The object case here is more complicated. The representation starts with the concept SUMMUM-GENUS:1, which is straightforward enough. SUMMUM-GENUS is the top concept on the tree, so that any sort of input will match. The initial SUMMUM-GENUS, however, is further constrained, so that what the representation really says is "any input such that ...". In fact, there are three constraints. First, is the concept NON-HOW-WHY-QUESTION. This is defined to be a question about what, where, when, who, whether, or how much; in short, anything but a question about how or why. This concept must be a secondary characterization rather than a primary one (which would appear in place of SUMMUM-GENUS:1) due to the way the concept tree is currently arranged. As it stands now, the question forms are split up under different concepts on the tree, so that their common superclass includes other concepts that we would not group under NON-HOW-WHY-QUESTION. This is a frequent occurrence when one is dealing with a tree: one set of relationships must be chosen to be primary, and the rest will then become secondary characterizations of concepts.

The second constraint on the object case says that the agent's goal is to know the answer to the question. This points up the goal-directed nature of the exchange. The third, rather involved, constraint on SUMMUM-GENUS:1 says that for the co-agent the object is

informationally nonspecific. This term means that on entry to the ASK-AND-ANSWER method, the co-agent cannot bind the SUMMUM-GENUS:1 to the question because he does not know it yet. (A variable may also be *existentially nonspecific*, i.e. it cannot be bound because the intended binding does not exist yet.) The -SELF specializer is present because specificity is treated as a property of variables, rather than their bindings; that is, (INFORMATIONALLY-NONSPECIFIC -SELF) is a property of SUMMUM-GENUS:1, not of a prospective binding. Note that in the second step of the method, after the question is asked, the co-agent is assumed to know the question, and an assertion is made about the new state of his knowledge base.

The semantic case specifications are followed by the steps in the method, separated by commas. (Other notation -- the use of THEN concepts connecting steps -- may be used when the linear ordering supplied by commas is insufficient.) The steps of the method are calls to other methods, and so they have associated input case assignments. To specify the values for these assignments, the semantic case names from the containing method are used as variables. For example, the first step is,

```
[(ASK OBJECT:)
  AGENT:: <-- AGENT:
  DESTINATION:: <-- CO-AGENT:]
```

When the Interpreter evaluates these case assignments, it finds that the agent of the step is the current value of the agent of ASK-AND-ANSWER, and the destination of the step is the current value of the co-agent of ASK-AND-ANSWER. OBJECT: in (ASK OBJECT:) evaluates to the object case setting of ASK-AND-ANSWER, which is the question to be asked. Thus, it is important not to confuse case *specifications* associated with methods and case *assignments* associated with calls.

The steps in the ASK-AND-ANSWER method have been discussed in Section 3.1 where Figure 2 gives a translation of the content. There remain only a few more points about notation. The SOME in the third step of Figure 6 indicates that the variable SUMMUM-GENUS:3 is nonspecific (in this example, informationally nonspecific), that is, it will not be bound until the FIND method has been carried out and the answer to the question has been found. The SOME specializer is a signal to Evaluate that no binding currently exists for this variable. Note that SOME is used to signal a nonspecific case *assignment*; specificity was mentioned above for the object of ASK-AND-ANSWER, but this was with regard to a case specification, rather than a case assignment.

Finally, we are in a position to return to the use of THE notation, for example in the fourth step, (STATE-AND-ACKNOWLEDGE (SUMMUM-GENUS:3 THE)). Specialization by THE is used to distinguish between concepts, so that SUMMUM-GENUS:3 and (SUMMUM-GENUS:3 THE) are, again, two different concepts. This time, however, the Interpreter has special mechanisms to relate the two forms, so that both variables evaluate to the same binding. Thus, once SUMMUM-GENUS:3 is bound to the answer in the third step, (SUMMUM-GENUS:3 THE) will evaluate to that answer. The reason for THE specializers is somewhat subtle, but it relates to the highly interrelated nature of the OWL-I knowledge base. The original SUMMUM-GENUS:3 has associated with it a set of modifiers, in particular that it is the answer to the question. Because of the way that reference lists are constructed, if the SUMMUM-GENUS:3 in the fourth step were not made into a separate concept, then that STATE-AND-ACKNOWLEDGE concept would end up on the reference list of SUMMUM-GENUS:3. This could lead to some confusion in the Interpreter, especially when it came time to

bind SUMMUM-GENUS:3 (whose binding was delayed because it is nonspecific). THE specializations were introduced to keep miscellaneous steps and relationships on a separate concept, distinguishing them from those modifiers that will be important in type checking for variable binding. Specialization by THE assures that no extraneous related concepts are retrieved when they are not wanted.

This covers the notation necessary to understand the ASK-AND-ANSWER example. From this it should be clear that OWL-I representations have different significance at different levels of the system, and that the notation, correspondingly, has more than one layer. LMS deals in concepts, specializations, and complexes which the user represents with parentheses, colons, square brackets, etc. The Interpreter, on the other hand, has a higher level, more semantic, point of view, and it deals with semantic cases, procedure calls, variables, and so forth.

Official Distribution List

Defense Documentation Center
Cameron Station
Alexandria, Va 22314 — 12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, Va 22217 — 2 copies

Office of Naval Research
Code 102IP
Arlington, Va 22217 — 6 copies

Office of Naval Research
Code 200
Arlington, Va 22217 — 1 copy

Office of Naval Research
Code 455
Arlington, Va 22217 — 1 copy

Office of Naval Research
Code 458
Arlington, Va 22217 — 1 copy

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, Ma 02210 — 1 copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, Il 60605 — 1 copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, Ca 91106 — 1 copy

New York Area Office
715 Broadway - 5th floor
New York, N. Y. 10003 — 1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375 — 6 copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380 — 1 copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, Ca 92152 — 1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Mathematics Department
Bethesda, Md 20084 — 1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350 — 1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division (OP-91T)
Office of Chief of Naval Operations
Washington, D. C. 20350 — 1 copy